Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 3 (2011) 1049–1054

Procedia Computer Science

www.elsevier.com/locate/procedia

WCIT-2010

A simple shuffle-based stable in-place merge algorithm

Mehmet Emin Dalkilic[a], Elif Acar[a], Gorkem Tokatli[a] *

[a]*International Computer Enstitute, Ege University, Izmir 35100, Turkey*

Abstract

Sorting is one of the most fundamental problems in computer science. Divide-and-conquer based sorting algorithms use successive merging to combine sorted arrays; therefore performance of merging is critical for these types of applications. The classic merge algorithm uses an extra *O(m+n)* memory for combining arrays of size *m* and *n*. Some improvements on merge algorithms include in-place methods which reduce or eliminate additional memory space, thus getting more practical value.

We present a new in-place, stable and shuffle-based merge algorithm which is simple to understand and program. The algorithm starts applying perfect shuffle on two sorted arrays. Then, using the knowledge that odd and even-indexed numbers are sorted among themselves, comparisons are made and then misplaced elements are relocated by applying successive inverse perfect shuffle and swap operations on blocks.

We have implemented and compared the new algorithm with the classical merge algorithm and the shuffle-based algorithm of Ellis and Markov (Comp. J. 1(2000)). Through experiments we have observed that the new algorithm exhibits linear run time behaviour and has dramatic performance improvement compared to the Ellis and Markov's algorithm [1].

Selection and/or peer-review under responsibility of the Guest Editor.

Keywords: Algorithms; merge; in-place merging; perfect shuffle; stable sorting

1. Introduction

Sorting numerical or alphabetical data is an essential and expensive operation for many computing applications. Merging two sorted lists into a single list is critical due to its usage in divide-and-conquer based sort algorithms. Mergesort is invented by John Von Neumann in 1945 and it has time complexity of *O(nlogn)* in the worst case [2]. Owing to its ideal worst case time complexity and not needing whole data at start-up, this algorithm is preferred by many applications.

Mergesort divides the unsorted list into two halves, and applies mergesort on each sub-list recursively, then combines the two lists back into one sorted list with *merge* algorithm. Merging is the operation of combining two sorted lists of sizes *m* and *n* into a single sorted sequence having *m+n* elements. Classical mergesort requires an additional *O(m+n)* memory space for merge operations. In this setting, merging takes linear time.

The need for extra memory space is a major drawback for the merge algorithm. To overcome this, large number of studies has been carried out for developing in-place merge algorithms. Kronrod described the first in-place, unstable merging with a fixed additional space [3]. He has used "block rearrangements" and "internal buffer" key

* Gorkem Tokatli. Tel.: +90-554-567-63-80

E-mail address: gorkem.tokatli@ege.edu.tr

1877-0509
doi:10.1016/j.procs.2010.12.172

techniques. Since Kronrod's algorithm was not stable, Horvath suggested a stable merge algorithm using Kronrod's key techniques [4]. Horvath's algorithm modifies keys, which is considered as a drawback. Pardo alleviated this drawback and derived an asymptotically optimal linear time algorithm without modifying keys [5].

Although these three algorithms have linear time complexity, they are not practically efficient and their structures are complex. Huang and Langston proposed a relatively practical in-place, unstable technique (not stated as a concrete algorithm) [6]. Later they have developed a stable version of this algorithm [7].

Geffert et al. presented two linear-time, in-place algorithms. Both algorithms perform at most $m(t+1)+n/2^t+o(m)$ comparisons where $m \leq n$ and $t = floor(\log_2(n/m))$. The first algorithm has semi-stable structure and has no more than $3(n+m)+o(m)$ element moves. The second one has a stable structure and has $5n+12m+o(m)$ moves [8]. Both algorithms have complex structures.

Possible improvements on upper and lower bounds in the study of Geffert et al. were left as an open problem. Chen [9] optimized the algorithm of Geffert et al. He simplified the algorithm with a cost of having worse asymptotic constants. This algorithm performs at most $m_1(t+1)+m_2/2^t+o(m_1+m_2)$ comparisons and $6m_2+7m_1+o(m_1+m_2)$ moves. Three of the proposed algorithms above are based on Mannila and Ukkonen's algorithm [10].

Ellis and Markov have described a novel and practical in-situ, stable merging algorithm where 'in-situ' means the use of no more than $O(\log^2 n)$ bits of extra memory for lists of size n [1]. In this algorithm, which is named *ShuffleMerge*, merge algorithm is performed by applying perfect shuffle permutation, that is exact interleaving of two equal length lists. Algorithm requires $\Omega(nlogn)$ in the worst case, and $O(nloglogn)$ in average, where n is the number of elements to be sorted. The algorithm stands out with its practicality and simplicity.

In this paper we introduce a new practical merge algorithm which is in-place, stable and shuffle-based. This algorithm has a simple structure. Through experiments it is observed that, the new algorithm exhibits linear run time behaviour and has dramatic performance improvement compared to the Ellis and Markov's algorithm.

The remaining sections of this paper are organized as follows: In Section 2 we describe the algorithm with a sample scenario. In Section 3 we give the test results of algorithm comparisons on several sizes of data sets and show that the new algorithm is faster than *ShuffleMerge* algorithm in all test results. Finally in Section 4, we conclude the paper stating the results, applications and future works.

2. The New Algorithm

The new algorithm consists of three basic functions: *In-shuffle*, *Inverse-shuffle* and *Block-swap*. *In-shuffle* interleaves two sequences by applying in-place perfect shuffle, and generates a new single sorted array that has i^{th} element from first sequence and $(i+1)^{th}$ element from the second sequence. An in-place perfect shuffle algorithm has been provided in study [11]. *Inverse-shuffle* is the inverse function of *In-shuffle* method. It decomposes one sequence and generates two sequences. The first sequence has the i^{th}, $(i+2)^{th}$, $(i+4)^{th}$,.. , $(i+2k)^{th}$ elements whereas the second sequence has $(i+1)^{th}$, $(i+3)^{th}$, $(i+5)^{th}$,...., $(i+2k+1)^{th}$ elements of the input array. This function can be implemented by applying steps of *In-shuffle* in reverse order. The last function *Block-swap*, swaps the two adjacent blocks. This operation is equivalent to applying n right cyclic shift in interval $[k, l]$ where k and l are the positions of the list. This method is also called *block rotation* [12].

The pseudo code of the new algorithm is given in Fig. 1. Variables used in the algorithm are as below:

curi: Index of element that will be correctly placed in current iteration.
N: Total number of elements in input arrays *A* and *B*.
nexti: Starting index of element(having different type) to be compared with *curi* in current iteration
blockStart*:* Starting index of *block structure*
blockSize: Total size of *block structure* that occurs in previous iteration
exNexti: *nexti* value of previous iteration, used to decide whether there exists a block structure.
count: Denotes whether a stopper is found in previous iteration. Also denotes the size of inverse shuffle interval.

The New Algorithm

```
1   Variables: curi,N,nexti,blockStart,blockSize,count,exNexti,C; //C[1,..N/2-1]=>type A,C[N/2,..N]=>type B
2   curi=1, nexti=2, count=0, blockSize=0;
3   In-shuffle(C,1,N/2,N); //Two sequences are shuffled together
4   while(curi < N-1)
5      exNexti=nexti; count=0;
6      if blockSize > 0
7         while ( C[curi] < C[nexti] AND curi < nexti ) //while curi is in its correct place
8            curi++
9         blockStart = curi ;
10     while (nexti ≤ N  AND  C[curi] > C[nexti] )     //If a stopper found from other type
11        count++; nexti=nexti+2
12     if (count > 0)
13        if (blockSize = 0)
14           C = Inverse-shuffle(C,curi,nexti-2)
15        else
16           if (exnexti ≠ curi)
17              C=Inverse-shuffle(C, exNexti-1,nexti-2)
18           Block-swap (C, blockStart, blockSize - 1, exNexti - 2, cnt)
19        curi = curi + count
20     //If no element greater than element in position curi
21     //therefore curi indexed element is in its correct place
22     else //if count = 0
23        if (exnexti != curi AND blockSize > 0) nexti = curi + 2; blockSize = 0
24        else if(blockSize>0)  nexti =curi + 1; curi = curi – 1
25        else  nexti=nexti+1;
26     curi=curi+1
27     blockSize = nexti – curi – 1
```

Fig. 1. Pseudo code of the New Algorithm

At the beginning of the algorithm, *In-shuffle* method is applied to the two sequences, combining them together. After performing *In-shuffle*, we set *curi* as the index of the first element, and *nexti* as the neighbour of *curi*. Initially *curi* is *1* and *nexti* is *2*. At the moment, the correct location of the element in *curi* will be found. This will be achieved by comparing it with elements from the other sequence. If element in *curi* is smaller than in *nexti*, then it is in correct place. Otherwise, we increment *nexti* to the next element of its sequence (applying *nexti=nexti+2*). Successive comparisons are made until an element in *nexti* greater than *curi* is found.

If we find an element in *nexti* greater than in *curi*, then we perform *Inverse-shuffle* algorithm to interval between *curi* and *nexti-2*. Otherwise, we should perform *Inverse-shuffle* algorithm to interval starting from *curi*, up to the end of the list. After the *Inverse-shuffle*, the element in *curi* is placed on its correct location. Therefore, the same procedures are applied with the right neighbor of that location.

After performing *Inverse-shuffle algorithm*, element in new *curi* may have adjacent neighbors which are from the same sequence. Such group is called *block*. In iteration, if we find that *Inverse-shuffle* should be performed and there exists a block structure, then two steps are applied. First, *Inverse-shuffle* is applied to the sequence in interval [*last block element, nexti-2*]. Then, the elements in [*blockStart, (blockStart+blockSize-2)*] interval are swapped with the adjacent elements of the other type, by using *Block-swap* function. These steps take place in 17^{th} and 18^{th} lines of the new algorithm in Fig 1. These operations can be seen in the sample scenario.

In Fig. 2, we have presented the execution steps of the new algorithm on a sample scenario. First input sequence is *A={4,50,60,65,67,70}*, second input sequence is *B={1,2,3,39,44,45}*. At the beginning, we apply *In-shuffle* algorithm to *A* and *B* arrays. The resulting sequence of this operation is shown in first iteration.

After *In-shuffle*, the algorithm starts comparing the first *B* type element with each of the *A* type elements from the beginning of the list. If the first *A* type element is greater than the *B* type element, then the *B* type element is in its correct place. Otherwise we continue comparing with next *A* type elements until we find an element that is greater (called stopper element), or we reach the end of the list.

In the first iteration we compare *1* with *4*, and find that *1<4* so *1* is in its correct place. Therefore we continue placing with the next element. In this case this element is *4*. In the second iteration we compare *4* with *B* type elements, *39* is found as the stopper element. Then *Inverse-shuffle* is applied to the elements between *4* and *3*.

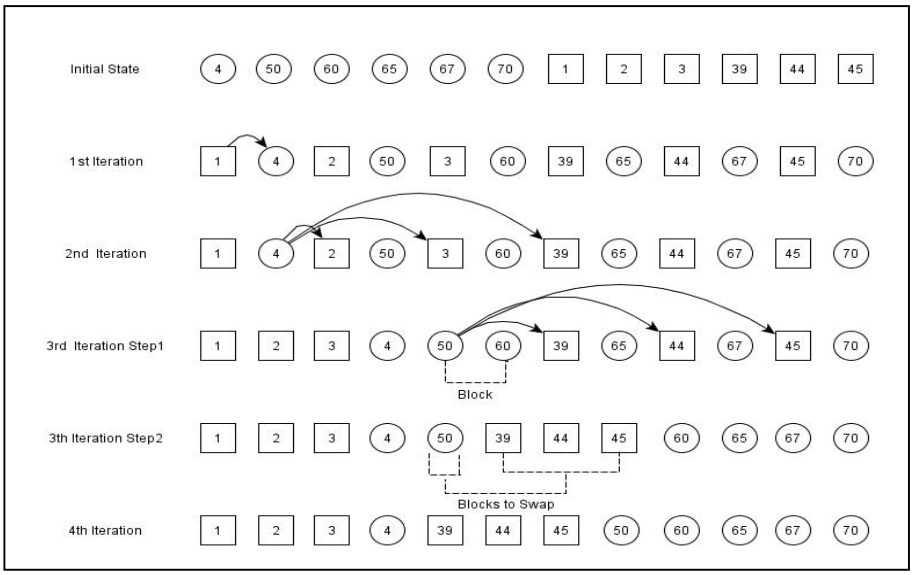

Fig. 2. Example running steps of the new algorithm

In third iteration, the new positions of elements between *4* and *3* can be seen. The first and the second iterations were similar, but in the third iteration there are two steps: *Inverse-shuffle* and *Block-swap*. *50* is compared with the next *B* type elements. There is an adjacent element having same type with *50*, forming a two-element block as seen in Fig. 1. In this iteration *50* is greater than all of *B* type elements, so *Inverse-shuffle* should be performed up to the end. *Inverse-shuffle* cannot be applied as before because of the existence of *block structure*. Instead, first an *Inverse shuffle* is applied between [*last element of block, end of list - 2*], which is between *60* and *45* in this example. Then, *Block-swap* is applied to the remaining block elements (*50* in this scenario) and the adjacent elements of the other type (*39*, *44* and *45* in this scenario). In the last iteration, the array is sorted and algorithm is finished.

3. Experimental Results

We have measured the performance of the new algorithm in comparison with *ShuffleMerge* [1] and classical merge algorithm. The algorithms are tested on a hardware platform with 3.00 GHz processor and 4 GB main memory. Java programming language is used for implementations.

In Table 1, running times of the three algorithms can be observed for different data set sizes and intervals indicated in each row. The numbers in the data set are generated as random 8-bit integers. For accuracy, these results are obtained by taking average of 10 experiments. Running speed is measured in milliseconds.

Table 1. Comparison of Merge Algorithms

Data Set Size	Class. Merge (ms)	ShuffleMerge (ms)	New Alg. Time (ms)	New Alg. / Class. Merge	ShuffleMerge / Class. Merge
2^{19}	5,684	14,955	17,906	3,150	2,631
2^{20}	4,617	24,566	20,548	4,450	5,320
2^{21}	9,648	40,970	32,607	3,379	4,246
2^{22}	19,076	84,146	56,704	2,972	4,411
2^{23}	41,771	216,043	117,628	2,816	5,172
2^{24}	72,537	892,792	197,418	2,722	12,308
2^{25}	150,114	1761,932	381,375	2,541	11,737
2^{26}	303,409	3754,007	791,032	2,607	12,373
2^{27}	548,513	8269,773	1391,930	2,538	15,077
2^{28}	1101,287	16360,430	2816,938	2,558	14,855
2^{29}	2274,074	36495,071	5748,587	2,527	16,048

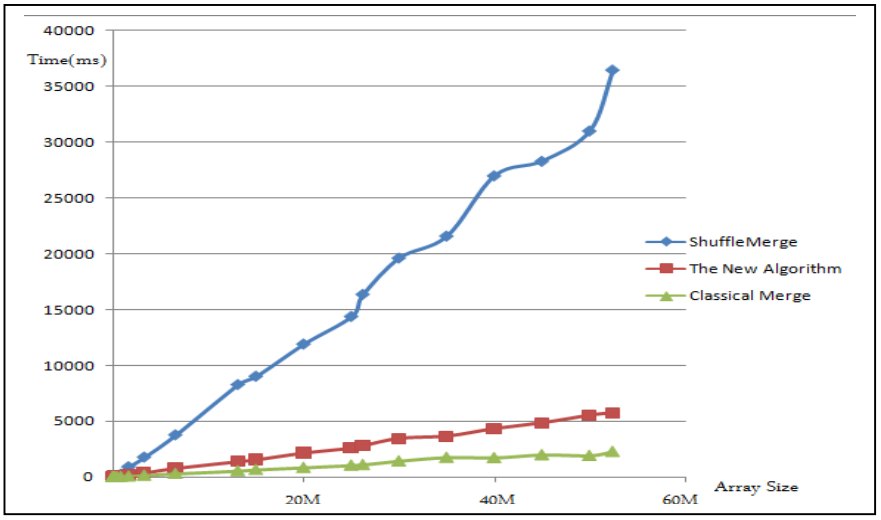

Fig. 3. Performance comparison of merge algorithms

The new algorithm is observed to have both practicality and linear time complexity, as seen on Fig. 3. The algorithm has a constant time ratio with the classical linear-time merge, as can be seen on Table 1. In compare, the *ShuffleMerge* algorithm has an increasing ratio with the classical merge, indicating its *O(nloglogn)* time complexity.

4. Conclusion

The need for extra *O(m+n)* space to merge arrays sized *m* and *n* is a key disadvantage for the classical merge algorithm. On the other hand, in-place merge efforts usually results in highly complex algorithms. In our attempt to produce a time-efficient simple in-place merge algorithm, we have used an in-place shuffle-based merging approach which resulted in an easy to understand and easily implementable algorithm also exhibiting linear time behavior. The performance improvements over Ellis and Markov's algorithm are substantial as presented on test results.

A formal analysis of the time complexity of the new algorithm is planned as a future work. In addition, we plan to improve the algorithm by developing a more efficient technique on finding inverse shuffle intervals. The algorithm can also be improved by optimizing the *Block-swap* algorithm.

Acknowledgements

One of the authors was supported by TUBITAK-BIDEB Ankara, TURKEY under grant #2211.

References

1. J. Ellis, M.Markov, In situ, Stable Merging by Way of the Perfect Shuffle, Comp. J 1 (2000) 43
2. D. E. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching, 2nd ed. Reading, MA: Addison-Wesley,1998
3. M. A. Kronrod, Optimal ordering algorithm without operational field, Soviet Math. Dokl. 10 (1969) 744-746
4. E. C. Horvath, Stable sorting in asymptotically optimal time and extra space. J. ACM, 25 (1978) 177-199.
5. L. T. Pardo, Stable sorting and merging with optimal space and time bounds. SIAM J. Comput., 6 (1977) 351-372
6. B.C. Huang and M. A. Langston, Practical in-place merging, Commun ACM, 31(1988) 348-352
7. B.C. Huang and M. A. Langston, Fast stable merging and sorting in constant extra space, Comp. J. 35 (1992) 643-650
8. V. Geffert, J. Katajainen, T. Passanen, Asymptotically efficient in-place merging, Theoret. Comput. Sci. 237 (2000) 159-181

9. J. Chen, Optimizing Stable in-Place Merging. Theoret. Comput. Sci. 302 (2003) 191-210
10. H. Mannila, E. Ukkonen, Simple linear time algorithm for in-situ merging, Inform. Process. Lett. 18 (1984) 203-208
11. P. Jain, A Simple In-Place Algorithm for In-Shuffle, CoRR abs/0805.1598, (2008)
12. P. Kim, A. Kutzner, On Optimal and Efficient in Place Merging SofSem, LNCS 3831, pp.350-359, Berlin, (2006)